(19)日本国特許庁（ＪＰ）　(12)公開特許公報（A）　(11)特許出願公開番号
特開2001－56764
（P2001－56764A）
(43)公開日　平成13年２月27日(2001.2.27)

| (51)Int.Cl.⁷ | 識別記号 | ＦＩ | ﾃｰﾏｺｰﾄﾞ(参考) |
|---|---|---|---|
| Ｇ０６Ｆ　9/45 | | Ｇ０６Ｆ　9/44 | ３２２Ｇ　５Ｂ０８１ |

審査請求　未請求　請求項の数８　ＯＬ　（全８頁）

(21)出願番号　特願平11－231649

(22)出願日　平成11年８月18日(1999.8.18)

(71)出願人　591112522
株式会社アクセス
東京都千代田区神田神保町１－64　神保協和ビル７階
(72)発明者　國澤　亮太
東京都千代田区神田神保町１丁目64番地　株式会社アクセス内
(74)代理人　100098350
弁理士　山野　睦彦

Ｆターム(参考)　5B081 DD01

(54)【発明の名称】　仮想計算機の実行方法および装置

(57)【要約】
【課題】中間コードからネイティブコードへの変換処理時間を軽減するとともに、比較的小さいメモリ容量の装置においても実行速度を総体的に向上させる。
【解決手段】中間コードの実行時に繰り返し実行されるループ部分を、該実行中に動的に検出し、当該ループ部分の中間コードをネイティブコードに変換する。中間コードの当該ループ部分については、ネイティブコードをＣＰＵにより直接実行する。より具体的には、中間コードをメモリ上にロードする際に、制御の流れが上に向かう命令を検出し、該命令の検出時に該命令を第１の特別の命令に書き換え、メモリ上の中間コードを実行する際、第１の特別の命令の実行時に制御の飛び先を確認し、再度同じ第１の特別の命令を実行したときに制御の飛び先が一致したならば、当該飛び先から飛び元を前記ループ部分であると判断し、該ループ部分のバイトコードをネイティブコードに変換する。

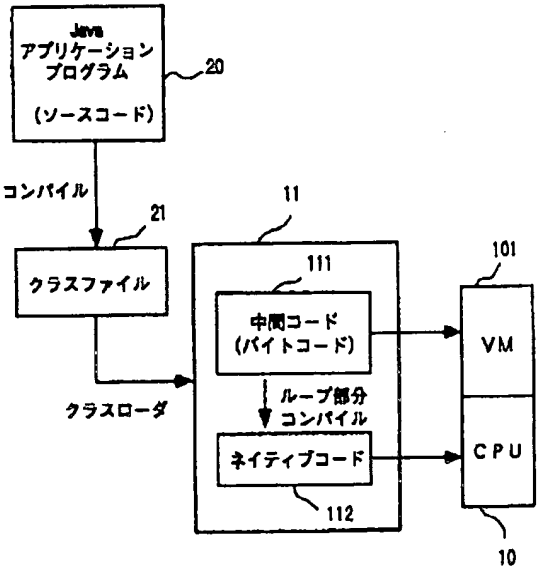


図１

【特許請求の範囲】
【請求項１】中間コードを実行する仮想計算機の実行方法であって、
中間コードの実行時に繰り返し実行されるループ部分を、該実行中に動的に検出するステップと、
当該ループ部分の中間コードをネイティブコードに変換するステップと、
前記中間コードの当該ループ部分については、前記ネイティブコードをＣＰＵにより直接実行するステップと、
を備えることを特徴とする仮想計算機の実行方法。
【請求項２】前記中間コードをメモリ上にロードする際に、制御の流れが上に向かう命令を検出し、該命令の検出時に該命令を第１の特別の命令に書き換えるステップと、
前記メモリ上の中間コードを実行する際、前記第１の特別の命令の実行時に制御の飛び先を確認し、再度同じ第１の特別の命令を実行したときに制御の飛び先が一致したならば、当該飛び先から飛び元を前記ループ部分であると判断し、該ループ部分の中間コードをネイティブコードに変換するステップと、
を有することを特徴とする請求項１記載の仮想計算機の実行方法。
【請求項３】前記第１の特別の命令に書き換えるステップを、中間コードのロード時に代えて実行時に行うことを特徴とする請求項２記載の仮想計算機の実行方法。
【請求項４】前記飛び先が一致した回数をカウントし、該カウント値が予め定めた値に達したとき、さらに、当該ループ部分のネイティブコードへの変換の可否を予め定めた判断基準で判断し、該判断基準が満足されたとき前記ネイティブコードへの変換を行うことを特徴とする請求項２または３記載の仮想計算機の実行方法。
【請求項５】前記判断基準が満足されなかったとき、当該第１の特別の命令を元の普通の命令に戻すことを特徴とする請求項４記載の仮想計算機の実行方法。
【請求項６】前記中間コードをメモリ上にロードする際に、前記「制御の流れが上に向かう命令」以外の制御の流れを変える命令を検出し、該命令の検出時に該命令を第２の特別の命令に書き換えるステップと、
前記メモリ上の中間コードを実行する際、前記第２の特別の命令の実行時に前記飛び先の情報を無効にするステップと、
を有することを特徴とする請求項２記載の仮想計算機の実行方法。
【請求項７】前記第２の特別の命令に書き換えるステップを、中間コードのロード時に代えて実行時に行うことを特徴とする請求項２記載の仮想計算機の実行方法。
【請求項８】中間コードを実行する仮想計算機の実行装置であって、
ネイティブコードを実行するＣＰＵと、
中間コードを記憶するメモリと、
該メモリ上に中間コードをロードする手段と、
該中間コードを解釈して実行する、前記ＣＰＵ上の仮想計算機手段と、
該仮想計算機手段による前記中間コードの実行時に繰り返し実行されるループ部分を、該実行中に動的に検出し、当該ループ部分の中間コードをネイティブコードに変換する手段とを備え、
前記中間コードの当該ループ部分については、前記ネイティブコードを前記ＣＰＵにより直接実行することを特徴とする仮想計算機の実行装置。
【発明の詳細な説明】
【０００１】
【発明の属する技術分野】本発明は、バイトコードと呼ばれる中間コードを実行する仮想計算機の実行方法および装置に関する。
【０００２】
【従来の技術】このような仮想計算機としては、ＪａｖａＶＭ（Ｊａｖａはサンマイクロシステムズ社の商標）が知られている。
【０００３】図１０に示すように、Ｊａｖａ言語で書かれたプログラム（ソースコード）はコンパイルされてバイトコードと呼ばれる中間コードに変換される。バイトコードは特定のハードウェアやＯＳ(Operating System)に依存せず、また比較的サイズが小さいため、ネットワーク上でプログラムを配布するのに適している。また、特定の装置への組み込み用のソフトウェアとしての利用にも適している。
【０００４】バイトコードは個々の計算機のＣＰＵが直接実行するのではなく、ソフトウェアで作られた計算エンジンである仮想計算機により実行される。なお、バイトコードに対して、ＣＰＵが直接実行できるコードはネイティブコードと呼ばれる。
【０００５】
【発明が解決しようとする課題】バイトコードはソフトウェアによって逐次解析され実行されるので、実行速度が遅いという難点がある。従来、この問題を解決する手法としてＪＩＴ (Just In Time)コンパイラというものが知られている。これは、バイトコードをネイティブコードに変換し（この処理もコンパイルと呼ばれる）、このネイティブコードを直接ＣＰＵが実行するものである。
【０００６】しかしながら、このＪＩＴには次のような問題があった。すなわち、第１に、このバイトコードからネイティブコードへのコンパイル処理に時間がかかる。第２に、バイトコードよりネイティブコードの方がサイズが大きいので、メモリ容量が比較的小さくＣＰＵが比較的低速である組み込み機器や民生機器では問題となる。
【０００７】本発明はこのような背景においてなされたものであり、その目的とするところは、中間コードから

ネイティブコードへの変換処理時間を軽減するとともに、比較的小さいメモリ容量の装置においても実行速度を総体的に向上させることができる仮想計算機の実行方法および装置を提供することにある。

【０００８】

【課題を解決するための手段】本発明による方法は、中間コードを実行する仮想計算機の実行方法であって、

【０００９】中間コードの実行時に繰り返し実行されるループ部分を、該実行中に動的に検出するステップと、当該ループ部分の中間コードをネイティブコードに変換するステップと、前記中間コードの当該ループ部分については、前記ネイティブコードをＣＰＵにより直接実行するステップとを備えることを特徴とする。

【００１０】本発明では、中間コードの実行時に動的にループ部分を検出することにより、確実にループ部分を検出できる。また、ループ部分のみをネイティブコードに変換するので、その変換に要する時間は比較的短く、かつ、ネイティブコードを保持するメモリ容量も比較的小さくて済む。

【００１１】より具体的には、前記中間コードをメモリ上にロードする際に、制御の流れが上に向かう命令を検出し、該命令の検出時に該命令を第１の特別の命令に書き換えるステップと、前記メモリ上の中間コードを実行する際、前記第１の特別の命令の実行時に制御の飛び先を確認し、再度同じ第１の特別の命令を実行したときに制御の飛び先が一致したならば、当該飛び先から飛び元を前記ループ部分であると判断し、該ループ部分の中間コードをネイティブコードに変換するステップとを有する。これにより、中間コードのロード時に、後の実行中におけるループ部分検出のための細工を中間コードに組み込んでおくことができる。

【００１２】但し、前記第１の特別の命令に書き換えるステップを、中間コードのロード時に代えて実行時に行うようにすることも可能である。

【００１３】前記飛び先が一致した回数をカウントし、該カウント値が予め定めた値に達したとき、さらに、当該ループ部分のネイティブコードへの変換の可否を予め定めた判断基準で判断し、該判断基準が満足されたとき前記ネイティブコードへの変換を行うようにしてもよい。これによりアプリケーションや用途に応じて、より適切なネイティブコードへの変換の条件を設定できる。

【００１４】前記判断基準が満足されなかったとき、当該第１の特別の命令を元の普通の命令に戻すことにより、当該ループについて以後、ネイティブコードへの変換に関する無駄な処理を行わないようにすることができる

【００１５】前記中間コードをメモリ上にロードする際に、前記「制御の流れが上に向かう命令」以外の制御の流れを変える命令を検出し、該命令の検出時に該命令を第２の特別の命令に書き換えるステップと、前記メモリ上の中間コードを実行する際、前記第２の特別の命令の実行時に前記飛び先の情報を無効にするステップとをさらに備えてもよい。これは、ループ処理が行われなかった、または中断された場合に相当し、この構成により上記と同様、以後の無駄な処理の実行を防止する。

【００１６】前記第２の特別の命令に書き換えるステップは、中間コードのロード時に代えて実行時に行うようにすることも可能である。

【００１７】上記方法を実行するための本発明による装置は、中間コードを実行する仮想計算機の実行装置であって、ネイティブコードを実行するＣＰＵと、中間コードを記憶するメモリと、該メモリ上に中間コードをロードする手段と、該中間コードを解釈して実行する、前記ＣＰＵ上の仮想計算機手段と、該仮想計算機手段による前記中間コードの実行時に繰り返し実行されるループ部分を、該実行中に動的に検出し、当該ループ部分の中間コードをネイティブコードに変換する手段とを備え、前記中間コードの当該ループ部分については、前記ネイティブコードを前記ＣＰＵにより直接実行することを特徴とする。

【００１８】

【発明の実施の形態】以下、本発明の実施の形態を詳細に説明する。

【００１９】図１は、本発明の仮想計算機の実行方法の概念図である。Ｊａｖａ言語等で記載されたアプリケーションプログラム（ソースコード）２０は、コンパイラにより中間コード（バイトコード）からなるクラスファイル２１に変換される。クラスとは、種々の機能を実現する複数の関数（メソッドと呼ばれる）を含むプログラムのかたまりであり、ひとつのアプリケーションに対して複数のクラスファイルが作成されうる。各クラスファイルの中間コードは、実行に先立ってメモリ１１内にロードされる。バイトコードの実行時に繰り返し実行されるループ部分を、プログラム実行時に検出するための細工を、このロード時に行うことができる。この具体例については後述する。

【００２０】メモリ１１上のバイトコード１１１は、ＣＰＵ１０上の仮想計算機（ＶＭ）１０１により逐次解釈され実行される。この実行中、上記細工に基づいてループ部分を検出したら、そのループ部分のバイトコードをネイティブコード１１２に変換してメモリ１１上に置く。このループ部分の実行時は、仮想計算機１０１によるバイトコードの実行に代わり、ＣＰＵ１０が当該ネイティブコードを直接実行する。後述するように、上記「細工」は実行中に行うことも可能である。

【００２１】通常、このような繰り返し部分は比較的小さいサイズなので、バイトコードをネイティブコードに変換する処理に要する時間は小さい。また、ループ部分のみに対応したネイティブコード１１２のサイズは小さくて済むので、ネイティブコード１１２のためにメモリ

１１の大容量化を招来することもない。

【００２２】図２は、本発明の方法が実行される装置のハードウェア構成を示すブロック図である。これは、一般的なコンピュータシステムであり、ＣＰＵ１０、メモリ１１、外部記憶装置（ハードディスク等）１２、キーボード等の入力装置１３、ディスプレイである表示装置１４、インターネットや外部装置との通信インタフェース部１５を有する。

【００２３】図３は、バイトコードをメモリ上にロードする際に行う処理のフローチャートである。

【００２４】まず、ひとつの関数のエントリを見つける（Ｓ１１）。ロードするバイトコードのプログラム中に複数の関数が存在する場合、未処理の関数がなくなるまで、以下の処理を繰り返す（Ｓ１２）。

【００２５】関数があれば、まず、その関数中の最初の命令を見つける（Ｓ１３）。この命令が「上に向かう命令」すなわち制御をプログラムの上方に戻す命令であるかを調べる（Ｓ１４，Ｙｅｓ）。これは、命令を実行する前にその記述に基づいて判断できる。「上に向かう命令」であった場合、この命令を「プロファイルを行う命令（第１の特別の命令）」に書き換える（Ｓ１７）。この第１の特別の命令は、「上に向かう命令」に対してプロファイルを行う機能を追加したものである。ここに、「プロファイル」とは、一般に、特定のベーシックブロックや、ソース中のある特定の行が、何回実行されたか、分岐があれば、どの方向に何回飛んだか、に関する情報であるが、ここでは、分岐のコードの場所、ならびに飛び先およびその回数である。「プロファイルを行う」とはこのような情報を確認し記憶することをいう。

【００２６】その後、後述するステップＳ１７へ進む。

【００２７】ステップＳ１４で当該命令が「上に向かう命令」ではなかった場合、「その他の制御の流れを変える命令」か否かを調べる（Ｓ１５）。この「その他の制御の流れを変える命令」は、ループ中の終了条件以外の場所でループから抜ける場合に相当する。このような命令であれば、当該命令を「プロファイルを無効にする命令」に書き換える（Ｓ１８）。この命令も、当該元の命令の機能に対して無効にする機能が追加されたものである。ここでプロファイルを無効にする理由は、「その他の制御の流れを変える命令」によって、先に「上に向かう命令」について「プロファイル」を作成していても、当該「上に向かう命令」でのループの繰り返しが実行されないことが判明したことになるからである。

【００２８】ステップＳ１５で当該命令が「その他の制御の流れを変える命令」でもなかった場合には、当該関数の終わりに達したかを調べ（Ｓ１６）、終わりに達していなければステップＳ１３に戻って上記処理を繰り返す。

【００２９】このようにして、次に説明するプログラム（バイトコード）の実行時に動的にループを見つけだす細工が、ロード時に行われたことになる。

【００３０】図４は、プログラム実行時の処理のフローチャートを示す。

【００３１】この処理では、まず、最初のバイトコード命令を取得し（Ｓ３１）、この命令が「プロファイルを行う命令」であるか否かを調べる（Ｓ３２）。そうであれば、プロファイルを作成する（Ｓ３３）。このプロファイルが先に記憶されているプロファイルと一致するか否かを調べる（Ｓ３４）。不一致ならば、今回得られた新たなプロファイルで先のプロファイルを上書きする（Ｓ３９）。最初にプロファイルを行う命令が出現したときには「先のプロファイル」は存在しないので、そのまま、今回のプロファイルを記憶する。

【００３２】ステップＳ３４で、今回のプロファイルが先のプロファイルと一致したら、同じ飛び元から同じ飛び先に飛んだと判定できるので、カウンタ（変数）をインクリメントする（Ｓ３５）。このカウンタの初期値は”０”としておく。このカウンタ値が予め定めた値（所定値）に達するまでは、ステップＳ３７，Ｓ３８を迂回してステップＳ４４へ進む。

【００３３】所定回数のループの繰り返しを待つのは、経験的に、あるいはそのアプリケーションの内容に基づいて、ループの繰り返しが当該所定回数に達するような場合はさらに繰り返しが続くであろう、かつ、繰り返しが当該所定回数に達しない場合には途中で繰り返しが停止または流れが変わる可能性が高いという想定に基づいている。したがって、カウンタの「所定値」は場合によって変わりうる。

【００３４】カウンタ値が所定値に達したら、当該ループ部分（飛び先から飛び元まで）のコンパイルを行う価値があるか否かを判断する（Ｓ３７）。この判断の内容については、後述する。コンパイルを行う価値がないと判断されれば、「プロファイルを行う命令」を元の普通の命令（上記追加機能のない命令）に戻す（Ｓ３８）。

【００３５】コンパイルを行う価値があると判断されれば、当該ループ部分のバイトコードをコンパイルしてネイティブコードを作成する（Ｓ４１）。ついで、このループの先頭のバイトコード命令を、ネイティブコードにジャンプする命令に書き換える（Ｓ４２）。以後、このループ部分についてはＣＰＵによりネイティブコードを直接実行する。ネイティブコードの実行後は、ループ内の最後のバイトコード命令の次のバイトコード命令に戻る。したがって、そのループの最後にある「プロファイルを行う命令」は迂回される。

【００３６】なお、ループを抜けた後も、当該ネイティブコードは保存されており、次に同じループに再度突入したときに同じネイティブコードが利用される。このように一度作成したネイティブコードを保存していくと、最終的にネイティブコードのメモリ容量が過大になりうるので、ネイティブコードを保存するメモリの最大容量

を予め定めておき、それを超えるようなネイティブコードの生成は行わないようにする。あるいは、ネイティブコード化された各ループの実行回数を記録管理しておき、メモリの最大容量を超えるような場合には、最も実行回数の低いループのネイティブコードを無効化するようにしてもよい。

【００３７】先のステップＳ３２において、命令が「プロファイルを行う命令」でない場合、「プロファイルを無効にする命令」であれば（Ｓ４３，Ｙｅｓ）、現在記憶されているプロファイル情報を無効にする（Ｓ４５）。

【００３８】「プロファイルを無効にする命令」でなければ、その命令を実行する（Ｓ４４）。先のステップＳ３８，Ｓ４５，Ｓ４２の後にも、このステップＳ４４に進む。ステップＳ４４の後は、最初のステップＳ３１に戻り、後続のバイトコード命令について上記処理を繰り返す。

【００３９】上記ステップＳ３７のコンパイルの実行可否を決めるための判断条件について説明する。このような判断条件の例として次の２つを挙げる。

（ａ）ループ内部の大きさとネイティブコードの読み出しのオーバーヘッドを比較する。ループ内部の大きさは、そのバイトコードの命令数で決まり、これに適当な値を掛けることにより、実際にループ１回の実行に要する時間を概算できる。ネイティブコードの読み出しのオーバーヘッドとは、ネイティブコードを実行するために余分に実行しなければならないコードで消費される時間に相当する。これは一定ではないが、一定と仮定する。「オーバーヘッド＋ネイティブコードをＣＰＵで実行する時間」が「ループ内部の大きさ」より小さい場合、ネイティブコードを実行した方が時間短縮できるので、コンパイルを行うべきと判断する。そうでなければ、バイトコードを仮想計算機で実行した方が速いか等しいので、コンパイルを行わないと判断する。なお、上記判断条件では、「ネイティブコードをＣＰＵで実行する時間」は０とみなしている。

（ｂ）ループ内部の大きさにジャンプした回数（ループを実行した回数）を掛けて、関数内部のコードの静的な大きさと比較する。ここで「ループ内部の大きさ」は上記の通りである。「ループ内部の大きさにジャンプした回数を掛けることにより、ループ内部のバイトコード命令を仮想計算機で実行した場合の消費時間が概算される。「関数内部のコードの静的な大きさ」は、「ループを１回だけ実行する時間＋ループ以外の部分の実行時間」を推定するものである。「ループ内部の大きさ」と「関数内部のコードの静的な大きさ」は、その関数内で当該ループの実行の占める時間の割合を表す。この割合が大きければ、コンパイルの対象にすべきと判断する。ある関数を実行する時間のうち、特定のループを実行する時間が大きければ、当該ループのコンパイルによる実行速度向上の寄与分が大きくなるからである。

【００４０】上記判断条件（ａ）（ｂ）はそれぞれ単独で採用しても、あるいは、ＡＮＤ条件で採用してもよい。また、ループのある程度の繰り返し回数が見込めるならば、実行時間短縮の効果が見込めるので、両条件の判断を行わない実施の形態も考えられる。すなわち、上記ステップＳ３７は削除し、カウンタ値が所定値に達したら直ちにネイティブコードへの変換を行うようにしてもよい。

【００４１】以下、図５～図８により、具体的なプログラム例を挙げて説明する。

【００４２】図５は、ベーシックブロックが１個の極めて簡単なソースプログラムの例を示す。この例は、整数ｉ，ａ，ｂについて、ａ，ｂの初期値をそれぞれ０と１０００とし、ｉ＝０から初めて、ａにａ＋ｂの値を代入し、ｉをインクリメントする処理を繰り返し、ｉの値が１０００に達したら、このループを抜けるものである。

【００４３】図６は、このソースプログラムをコンパイルして得られたバイトコードの例を示す。この例では、変数ｉはvar[8]、ａはvar[9]、ｂはvar[10]に割り付けられている。変数ａ，ｂの初期化のコードは省略してある。図６の番地３４から番地５２の前までがループ部分であり、これがネイティブコードへのコンパイルの対象となる。

【００４４】これらのバイトコード列のロード時に、番地４９に上の番地３４に向かう命令があるため、図７に示すように、この番地４９の命令が前記「プロファイルを行う命令」に書き換えられる。また、「プロファイルを無効にする命令」は存在しない。

【００４５】この図７のバイトコード列のプログラムを仮想計算機で実行する際、番地３４から番地５２の前までのループ部分を所定回数繰り返した時点でコンパイルにより当該部分がネイティブコードに変換され、図８に示すように、ループの先頭の番地３４の命令が「ネイティブコードに飛ぶための命令」に書き換えられる。ネイティブコードの具体例については、使用するＣＰＵにより異なり、また、容量も大きいので、ここでは特に例示はしない。

【００４６】図９は、本発明による他の実施の形態におけるプログラム実行時の処理フローを示す。先の実施の形態では、バイトコードのロード時（図３）に、ループ部分検出のための細工を行ったが、この実施の形態ではプログラム実行処理の中に図３の処理を含めたものである。このようにすることにより、プログラム実行開始までに要する時間を短縮することができる。これは、起動が速いことが望まれる一方、実行時にかかる時間はそれほど気にしない場合（例えば、インタラクティブな処理の場合）に適している。これに対して、先の実施の形態は、実行開始までに時間がかかっても実行時の処理時間は極力短縮したいような場合（例えば、リアルタイム処

理の場合）に適している。

【００４７】図９において、具体的には、全ステップＳ５１～Ｓ７１のうち、Ｓ５４，Ｓ５５，Ｓ５８，Ｓ６０が、先の実施の形態におけるプログラムロード時の処理ステップである。ステップＳ５２～Ｓ５６は順不同である。また、ステップＳ５７のエラー処理では、あり得ない命令の検知時にその命令をスキップするか、あるいは仮想計算機を異常終了するものである。このステップは図４の実行処理には示さなかったが、図４の処理において含めてもよい。

【００４８】以上、本発明の好適な実施の形態について説明したが、本発明の要旨を逸脱することなく、種々の変形・変更を行うことが可能である。例えば、上記ではＪａｖａアプリケーションの処理について説明したが、いわゆるアプレットと呼ばれるプログラムを処理するＷｅｂブラウザに内蔵された仮想計算機にも本発明は同様に適用可能である。

【００４９】

【発明の効果】本発明の仮想計算機の実行方法および装置によれば、中間コードからネイティブコードへの変換処理時間を軽減するとともに、比較的小さいメモリ容量の装置においても実行速度を総体的に向上させることができる。

【図面の簡単な説明】

【図１】本発明の仮想計算機の実行方法の概念図である。

【図２】本発明の方法が実行される装置のハードウェア構成を示すブロック図である。

【図３】本発明の実施の形態において、バイトコードをメモリ上にロードする際に行う処理のフローチャートである。

【図４】本発明の実施の形態におけるプログラム実行時の処理のフローチャートを示すフローチャートである。

【図５】ベーシックブロックが１個の極めて簡単なソースプログラムの例を示す図である。

【図６】図５ののソースプログラムをコンパイルして得られたバイトコードの例を示す図である。

【図７】図６のバイトコード列の中の命令が「プロファイルを行う命令」に書き換えられた様子を示す図である。

【図８】図７のバイトコード列中のループの先頭の命令が「ネイティブコードに飛ぶための命令」に書き換えられた様子を示す図である。

【図９】本発明の他の実施の形態におけるプログラム実行時の処理のフローチャートを示すフローチャートである。

【図１０】Ｊａｖａ言語で書かれたプログラム（ソースコード）はコンパイルされてバイトコードと呼ばれる中間コードに変換される様子を示す概念図である。

【符号の説明】

１０　ＣＰＵ
１１　メモリ
２０　アプリケーションプログラム（ソースコード）
２１　クラスファイル
１０１　仮想計算機（ＶＭ）
１１１　中間コード（バイトコード）
１１２　ネイティブコード

【図１】

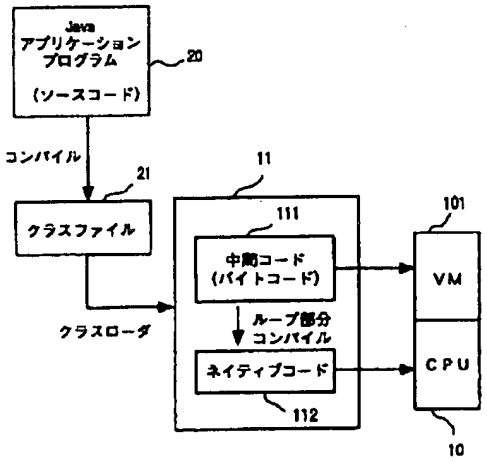


図１

【図２】

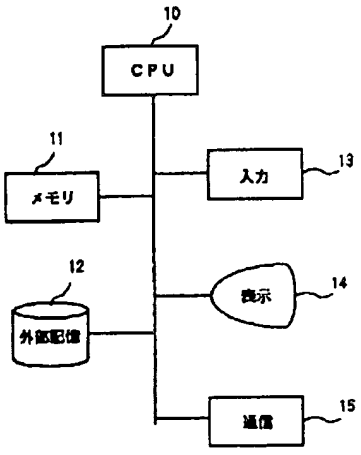


図２

【図３】

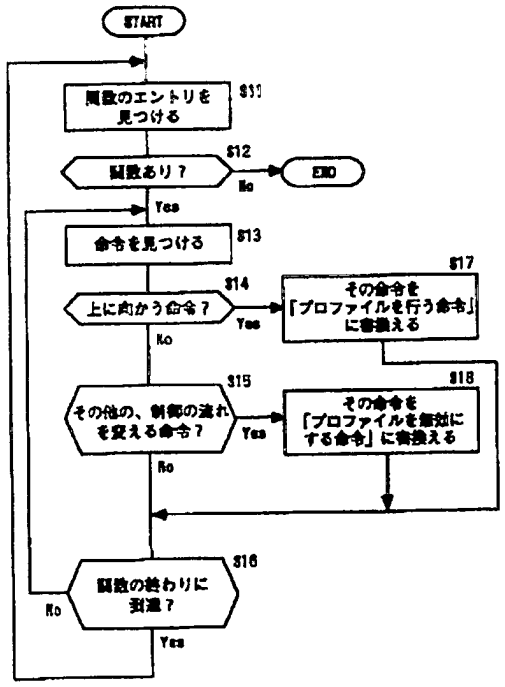


図３　バイトコードロード時の処理

【図４】

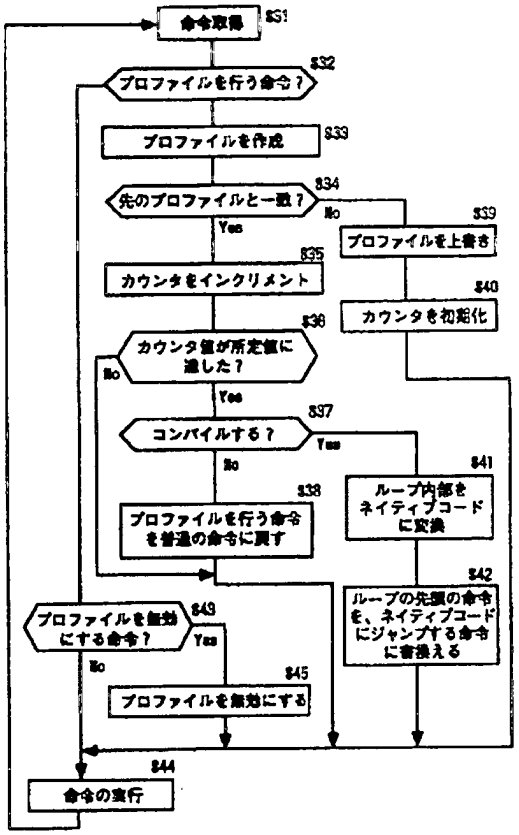


図４　プログラム実行時の処理

【図５】

```
int i, a = 0, b = 1000;
for (i = 0; i < 1000; i++) {a += b;}
```

図５　ソース（java）

【図６】

```
31 iconst_0      /* push 0;*/
32 istore 8      /* pop to var[8]; (i = 0) */
34 iload 9       /* push var[9]; */
36 iload 10      /* push var[10]; */
38 iadd          /* add; */
39 istore 9      /* pop to var[9]; (a = a + b) */
41 iinc 8 1      /* var[8] += 1; (i += 1) */
44 iload 8       /* push var[8] */
46 sipush 1000   /* push 1000; */
49 if_icmplt 34 /* t = (sp[1] < sp[0]); pop; pop; if t goto 34; */
52 ....
```

図６　コンパイルされた後のバイトコード

【図７】

```
31 iconst_0     /* push 0;*/
32 istore 8     /* pop to var[8]; (i = 0) */
34 iload 9      /* push var[9]; */
36 iload 10     /* push var[10]; */
38 iadd         /* add; */
39 istore 9     /* pop to var[9]; (a = a + b) */
41 iinc 8 1     /* var[8] += 1; (i += 1) */
44 iload 8      /* push var[8] */
46 sipush 1000  /* push 1000; */
49 isdep2 34     // プロファイルを行なう icmplt/
52 ....
```

図7　変換後のバイトコード

【図８】

```
31 iconst_0     /* push 0;*/
32 istore 8     /* pop to var[8]; (i = 0) */
34 isdep1        // ネイティブコードに飛ぶために使用
36 iload 10     /* push var[10]; */
38 iadd         /* add; */
39 istore 9     /* pop to var[9]; (a = a + b) */
41 iinc 8 1     /* var[8] += 1; (i += 1) */
44 iload 8      /* push var[8] */
46 sipush 1000  /* push 1000; */
49 isdep2 34     // プロファイルを行なう icmplt/
52 ....
```

図8　実行時に変換されたバイトコード

【図９】

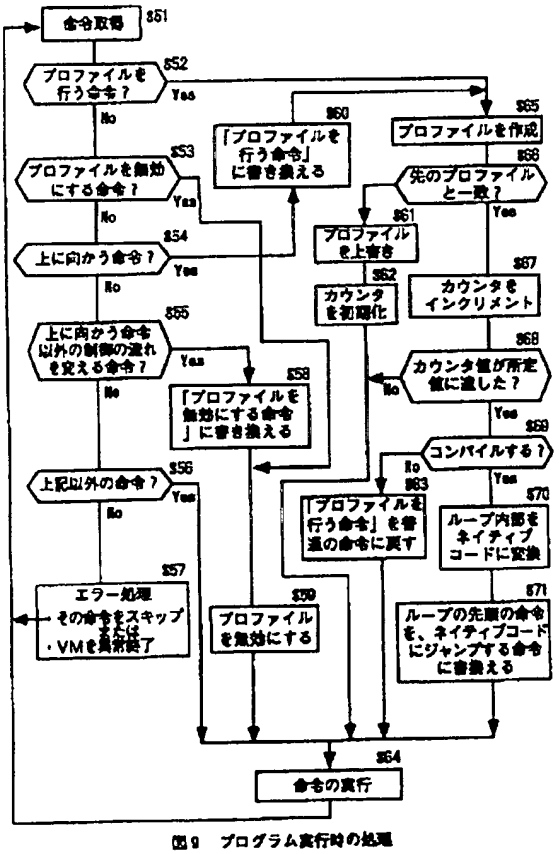


図9　プログラム実行時の処理

【図１０】

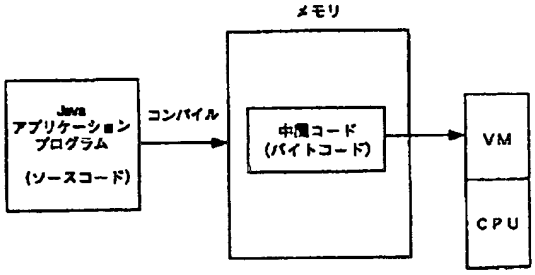


図10